Numerical Methods
ITGS219

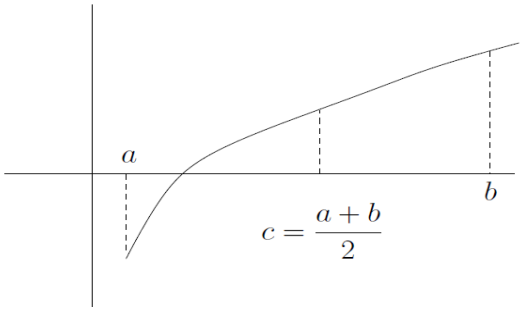
Lecture 6

Root Finding

By: Zahra A. Elashaal

Root Finding (Bisection Method)

- In the previous example we used an initial estimate for a root: here we shall use a bracketing **interval**.
- Our initial assumption is that this **interval** contains a **single root**. In order to check this hypothesis we shall build in checks at each stage.
- If the initial interval is between $x = a$ and $x = b$ we know that $f(a)$ and $f(b)$ are *of different signs (for the interval to contain a single root): in which* case their product must be negative.
- As the name of the method suggests we bisect the interval and define the point $c = (b+a)/2$.
- *We can now evaluate the* function to obtain $f(c)$.
- *This must either be positive or negative (or if we are very lucky zero).*
- *In which case $f(c)$ will have the same sign as* either $f(a)$ or $f(b)$.
- *The new interval can then be defined as c and whichever end of the previous interval represents a change in sign of the function.*



Root Finding (Bisection Method

- A simple version of a code to use this method in an interval [*a*, b] would be

```
a = 1; b = 5;
for j = 1:10
  c = (a+b)/2;
  if f(c)*f(a) > 0
     a  = c;
   else
     b = c;
  end
end
```

If we wish to be slightly more careful we would use the codes

```
% bisect.m
function [answer, iflag] = bisect(fun, a, b)
global tolerance maxits
iflag = 0;
iterations = 0 ;
f_a = feval(fun,a);
f_b = feval(fun,b);
 while ((f_a*f_b<0) & iterations < maxits) & (b-a)>tolerance
     iterations = iterations + 1 ;
     c = (b+a)/2;
     f_c = feval(fun,c);
     if f_c*f_a<0
           b=c; f_b = f_c;
     elseif f_b*f_c < 0
           a=c; f_a = f_c;
     else
          iflag = 1; answer = c;
     end
 end
switch iterations
   case maxits
        iflag = -1; answer = NaN;
   case 0
        iflag = -2; answer = NaN;
   otherwise
        iflag = iterations; answer = c;
end
```

- where in this case *a = 1*, *b = 5* and the function *f(x)* is defined in a *routine f.m.*
- This method is perfectly adequate if you know that the method will *converge* in 10 steps, the interval [*a*, b] *actually contains a root and no future iterations* actually coincide with a root

Root Finding (Bisection Method Cont.)

```
% func.m
function [f] = func(x)
f = x-2*sin(x.^2);
```

```
% mbisect.m
global tolerance maxits
tolerance = 1e-4;
maxits = 30;
xlower = 0.4;
xupper = 0.6;
[root,iflag] = bisect('func', xlower, xupper);
switch iflag
  case -1
    disp('Root finding failed')
 case -2
    disp('Initial range does not only contain one root')
 otherwise
    disp([' Root = ' num2str(root) ...' found in '
    num2str(iflag) ' iterations'])
end
```

- This method is guaranteed to work provided only one root is in the relevant interval and the function is *continuous*.
- It may work if there are three roots but this is not recommended: in fact it appears to work provided there are an odd number of roots, because each iteration may remove a number of roots which is guaranteed to be even.
- We note that the length of the interval after *n iterations* is $(b-a)/2^n$. Hence if we wish to know the *root* to within a *given tolerance* we can work out **how many iterations we need** to perform.
- So that, if the required *tolerance* is ***e***, we find that:

Number of iterations we need is: $n > \frac{1}{\ln 2} \ln\left(\frac{b-a}{\epsilon}\right)$.

Example 4.2 To determine a root of a continuous function *f(x)* between [0,1], given that *f(0)f(1) < 0* to within 1×10^{-4} requires $n > \ln(10^4)/\ln 2 \approx 13.28$: in other words ***14*** *iterations.*

Root Finding(Bisection Method Cont ...)

Bisection Method Example

Determine the root of the given equation $x^2-3 = 0$ for x ∈ [1,2]

- Given: $x^2-3 = 0$ => Let $f(x) = x^2-3$
- Now, find the value of $f(x)$ at a=1 and b=2.

$$f(x=1) = 1^2-3 = 1 - 3 = -2 < 0$$

$$f(x=2) = 2^2-3 = 4 - 3 = 1 > 0$$

- The given function is continuous, and the root lies in the interval [1, 2].
- Let "c" be the midpoint of the interval. => ***c = (a+b)/2***

c = (1+2)/2 , c =3 / 2 , c = 1.5

- Therefore, the value of the function at "c" is

$$f(c) = f(1.5) = (1.5)^2-3 = 2.25 - 3 = -0.75 < 0$$

- If f(c) < 0, let a = c. And If f(c) > 0, let b = c.
- f(c) is negative, so a is replaced with c = 1.5 for the next iterations.

The iterations for the given function are:

Iteration	a	b	c	*f(a)*	*f(b)*	*f(c)*
1	1	2	1.5	-2	1	-0.75
2	1.5	2	1.75	-0.75	1	0.062
3	1.5	1.75	1.625	-0.75	0.0625	-0.359
4	1.625	1.75	1.6875	-0.3594	0.0625	-0.1523
5	1.6875	1.75	1.7188	-01523	0.0625	-0.0457
6	1.7188	1.75	1.7344	-0.0457	0.0625	0.0081
7	1.7188	1.7344	1.7266	-0.0457	0.0081	-0.0189

- So, at the seventh iteration, we get the final interval [1.7266, 1.7344]
- Hence, 1.7344 is the approximated solution.

Root Finding(Bisection Method Cont ...)

Algorithm

Bisection method Steps (Rule)	
Step-1:	Find points a and b such that $a < b$ and $f(a) \cdot f(b) < 0$.
Step-2:	Take the interval $[a, b]$ and find next value $x_0 = \frac{a+b}{2}$
Step-3:	If $f(x_0) = 0$ then x_0 is an exact root, else if $f(a) \cdot f(x_0) < 0$ then $b = x_0$, else if $f(x_0) \cdot f(b) < 0$ then $a = x_0$.
Step-4:	Repeat steps 2 & 3 until $f(x_i) = 0$ or $\lvert f(x_i) \rvert \le$ Accuracy

Example-2 By using Bisection method find the root for $f(x)=x^3-x-1$ in the interval [1,2] and with acceptance error =0.0005

Solution:

Here $x^3-x-1=0$ => Let $f(x)=x^3-x-1$

Here $f(1) = -1 < 0$ and $f(2) = 5 > 0$ so the root lies between 1 and 2

as $e=0.0005$ then number of iterations is:

$$n > \frac{1}{\ln 2} \ln\left(\frac{b-a}{\epsilon}\right).$$

n > 1/ln2 * ln (1/.0005)

> 1.44*7.6 > 10.94 ∴ n = 11

$c = (a+b)/2$ = (1+2)/2 = 1.5 > 0

	a	b	c
x	0	2	1.5
f(x)	-1	5	0.875

Root Finding(Bisection Method Cont …)

Example-2 By using Bisection method find the root for $f(x)=x^3-x-1$ in the interval [1,2]

n	a	$f(a)$	b	$f(b)$	$c=\frac{a+b}{2}$	$f(c)$	Update
1	1	-1	2	5	1.5	0.875	$b=c$
2	1	-1	1.5	0.875	1.25	-0.29688	$a=c$
3	1.25	-0.29688	1.5	0.875	1.375	0.22461	$b=c$
4	1.25	-0.29688	1.375	0.22461	1.3125	-0.05151	$a=c$
5	1.3125	-0.05151	1.375	0.22461	1.34375	0.08261	$b=c$
6	1.3125	-0.05151	1.34375	0.08261	1.32812	0.01458	$b=c$
7	1.3125	-0.05151	1.32812	0.01458	1.32031	-0.01871	$a=c$
8	1.32031	-0.01871	1.32812	0.01458	1.32422	-0.00213	$a=c$
9	1.32422	-0.00213	1.32812	0.01458	1.32617	0.00621	$b=c$
10	1.32422	-0.00213	1.32617	0.00621	1.3252	0.00204	$b=c$
11	1.32422	-0.00213	1.3252	0.00204	1.32471	-0.00005	$a=c$

Approximate root of the equation $x^3-x-1=0$ using Bisection method is 1.32471

Root Finding(Newton–Raphson and Secant Methods)

- The central premise for both methods is that the function is locally linear and the next iteration for the
- required value can be attained via linear extrapolation (or interpolation).
- **Derivation of the Newton–Raphson Method**
- We will start using a Taylor series to derive the Newton–Raphson technique2. We assume the current guess is *x and this is incorrect by an amount h, so that x + h is the required value. It now remains for us to determine h or at least* find an approximation to it. The Taylor expansion for the function *f(x) at the* point *x + h is given by*

$$f(x+h) = f(x) + hf'(x) + O(h^2).$$

- This can be interpreted as: the value of the function at *x + h is equal to the* value of the function at *x plus the gradient times the distance between the* points. This can be considered to include further terms; at the moment we are fitting a straight line.
- We now note that *x + h is supposedly the* actual root so *f(x+h) = 0, and discarding the higher-order terms we find that*

$$h \approx -\frac{f(x)}{f'(x)}.$$

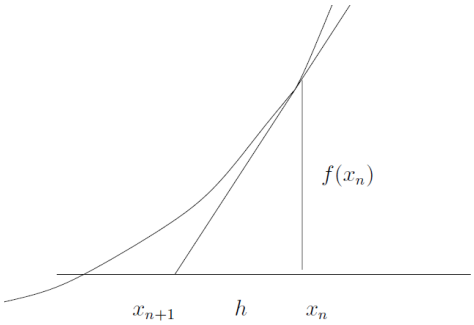
Root Finding(Newton–Raphson and Secant Methods)

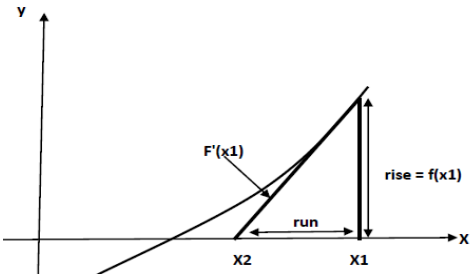
This presumes we are close to the actual root, and consequently we can discard the terms proportional to *h2 since these should be smaller than those* proportional to *h*.

- This allows us to construct the iterative scheme

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n = 0, 1, 2, \cdots .$$

- This method can also be derived using geometric arguments. In these derivations the function is taken to be approximated by a straight line in order to determine the next point.





F'(x1) = rise / run

F'(x1) = f(x1)/run

Run = f(x1)/f'(x1)

X1-x2 = f(x1)/f'(x1)

X2= x1-(f(x1)/f'(x1)) ----- newton's formula.

Root Finding(Newton–Raphson and Secant Methods)

At the moment we shall presume that we have two routines *func.m* and func *prime.m* which give us the function and its derivative.

- For ease let us consider the function *f(x) = x − 2 sinx², so f(x) = 1 − 4x cos x² (using the* chain rule). The code *func.m* is given on page 115 and the other one is:

```
% func_prime.m
function [value] = func_prime(x)
value = 1 - 4*x.*cos(x.^2);
```

- Notice that we have used the dot operators, even though this routine is only ever likely to be called in this context using a scalar. This permits the routine to be used from other codes in a portable fashion.
- This can be coded simply using

```
x = 1;
for j = 1:10
    x = x - func(x)/func_prime(x);
end
```

- where we have set the initial guess to be *x = 1 and supposed that the method* will converge in 10 iterations.

Root Finding(Newton–Raphson and Secant Methods)

We now give a more robust code to perform the iterations

- Hopefully you can see the difference between the two codes and see that ultimately
- the second version is more useful. By entering the values 0.*1, 0.6 and*
- 1.5 *we can obtain the three roots we are concerned with. Notice we have increased*
- the number of digits printed in the answer to 10 using the form of the
- MATLAB command num2str which accepts two arguments.

```
% Newton_Raphson.m
x = input('Starting guess :');
tolerance = 1e-8;
iterations = 0;
while (iterations<30) & (abs(func(x))>tolerance)
     x = x-func(x)/func_prime(x);
     iterations = iterations + 1;
end
if iterations==30
      disp('No root found')
else
  disp([' Root = ' num2str(x,10) ' found in ' ...
   int2str(iterations) ' iterations.'])
end
```

Root Finding(Newton–Raphson and Secant Methods)

We will discuss the **secant technique**

- This method starts with two values of *x*, namely $x = x_0$ and $x = x_1$.
- *A* straight line is drawn between the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$, which has the *equation*

$$\frac{y - f(x_0)}{f(x_1) - f(x_0)} = \frac{x - x_0}{x_1 - x_0}.$$

- We wish to find the value of x (x_2 *say)* for which $y = 0$ which is given by

$$x_2 = x_0 - f(x_0)\frac{x_1 - x_0}{f(x_1) - f(x_0)}.$$

- We then move on so that we are going to use $x = x_1$ and $x = x_2$ as the next two points.
- This is coded to give:

```
% Secant.m
x0 = input('Starting guess point 1 :');
x1 = input('Starting guess point 2 :');
tolerance = 1e-8;
iterations = 0;
while (iterations<30) & (abs(func(x1))>tolerance)
    iterations = iterations + 1 ;
    f0 = func(x0);
    f1 = func(x1);
    x2 = x0-f0*(x1-x0)/(f1-f0);
    x0 = x1;
    x1 = x2;
end
if iterations==30
    disp('No root found')
else
    disp([' Root = ' num2str(x1,10) ' found in ' ...
    num2str(iterations) ' iterations.'])
end
```

Root Finding(Newton–Raphson and Secant Methods)

- This method works far better if the two initial points are on opposite sides of the root.
- In the above method we have merely chosen to proceed to use x_1 *and* the newly attained point x_2*: however we could equally have chosen* x_0 *and* x_2.
- In order to determine which we should use we require that the function *changes sign* between the two ends of the interval.
- This is done by changing the lines where the next interval is chosen.

$$x_2 = x_0 - f(x_0)\frac{x_1 - x_0}{f(x_1) - f(x_0)}.$$

```
%False_Position.m
x0 = input('Starting guess point 1 :');
x1 = input('Starting guess point 2 :');
x2 = x0;
tol = 1e-8;
iters = 0;
while ((iters<30) & (abs(func(x2))>tol))|(iters==0)
      iterations = iterations + 1 ;
      f0 = func(x0);
      f1 = func(x1);
      x2 = x0-f0*(x1-x0)/(f1-f0);
      if func(x2)*f0 < 0
            x1 = x2;
      else
            x0 = x2;
      end
end
if iters==30
      disp('No root found')
else
      disp(['Root = ' num2str(x2,10) ' found in ' ...
      num2str(iters) ' iters.'])
end
```

Root Finding(Newton–Raphson and Secant Methods)

Example 4.3 consider the function $f(x) = x^m - a$ where $a>0$. The roots of this equation are $\sqrt[m]{a}$ or, written another way, $a^{1/m}$.

For this equation we can write the Newton–Raphson scheme as:

$$x_{n+1} = x_n - \frac{x_n^m - a}{m x_n^{m-1}},$$

which can be simplified to yield:

$$x_{n+1} = x_n\left(1 - \frac{1}{m}\right) + \frac{a}{m x_n^{m-1}}.$$

Notice this is exactly the map solved in the example on page 91, with a = 3 and m = 2.

Example 4.4 Using the Newton–Raphson routine determine the zero of the *function*

$$f(x) = e^x - e^{-2x} + 1.$$

This can be done by setting up the functions

```
function [value] = func(x)
value = exp(x) -exp(-2*x)+1;
```

```
function [value] = func_prime(x)
value = exp(x)+2*exp(-2*x);
```

This yields the result

```
>> Newton_Raphson
Starting guess :-2
Root = -0.2811995743
found in 8 iterations.
```

(with a tolerance of 1×10^{-10}). In fact we can solve this equation by introducing the variable $Y = e^x$, and noting that Y is never zero. We can rewrite f(x) as

$$f(x) = \frac{1}{Y^2}\,(Y^3 - 1 + Y^2)$$

real root is found using the Matlab code *co=[1 1 0 -1]; roots(co). Then to retrieve the root of the equation we use the fact that x = lnY.*

Matlab Routines for Finding Zeros

Roots of a Polynomial

- As we have seen Matlab has a specific command for finding the roots of a polynomial, namely ***roots***.
- The *coefficients* of the polynomial are placed in a vector ***c*** and the routine returns the corresponding roots.
- It is very simple to use, but care is need when entering the coefficients.

Example 4.6 Find the roots of the quantic equation $f(x) = x^5 - x^4 + x^3 + 2x^2 - 1$.

This is accomplished using the code:

```
c = [1  -1   1  2  0  -1];
roots(c)
```

There are a couple of things to note from this example:

- The polynomial's coefficients are listed starting with the one corresponding to the **largest power**.
- It is crucial that **zeros** are included in the sequence where necessary (in the above we have included the zero times *x term).*
- As a simple check, a polynomial of **order** or **degree *p*** *has* ***p + 1 coefficients***, and it *will have* ***p*** **roots**.
- So as long as the coefficients of the polynomial are real, the roots will be real or occur in complex conjugate pairs.

MATLAB Routines for Finding Zeros (Cont.)

The Command *fzero*

The Matlab command *fzero* is very powerful. It actually chooses which scheme should be used to solve a given problem. Let us try the form of the command.

```
fzero ('func', 0.4, optimset('disp','iter'))
```

This produces →

The command uses many different forms but here we are looking for a root of the function defined in *func.m* near the value $x = 0.4$ and the *options are set* to display *iterations*.

```
 Func-count      x          f(x)         Procedure
     1          0.4      0.0813636       initial
     2       0.388686    0.0876803       search
     3       0.411314    0.0745675       search
     4          0.384     0.0901556       search
     5          0.416      0.071613       search
     6       0.377373    0.0935142       search
     7       0.422627     0.067296       search
     8          0.368     0.0979791       search
     9          0.432     0.0609148       search
    10       0.354745     0.103721       search
    11       0.445255     0.0513434       search
    12          0.336     0.110687       search
    13          0.464     0.0367268       search
    14        0.30949      0.118215       search
    15        0.49051     0.0139394       search
    16          0.272     0.124167       search
    17          0.528    -0.0223736       search

   Looking for a zero in the interval [0.272, 0.528]

    18       0.488914     0.0153797       interpolation
    19       0.504837   0.000616579       interpolation
    20       0.505484   -1.5963e-06       interpolation
    21       0.505482   1.80934e-09       interpolation
    22       0.505482   5.32907e-15       interpolation
    23       0.505482             0       interpolation
Zero found in the interval: [0.272, 0.528].

ans =

   0.50548227233930
```

MATLAB Routines for Finding Zeros (Cont..)

Example 4.7 Determine the zero of the function $f(x) = x - x^2 \sin x$ nearest to $x = 1$.

We need to set up the code

```
function [val] = myfunc(x)
val = x-x.^2.*sin(x);
```

and then use the inline command ***fzero****('myfunc',1).n This gives*

```
>> fzero('myfunc',1)
    Zero found in the interval: [0.84, 1.16].
    ans =    1.1142
```

HomeWork tasks Page 130 - 132 from Task 4.2 To Task 4.13